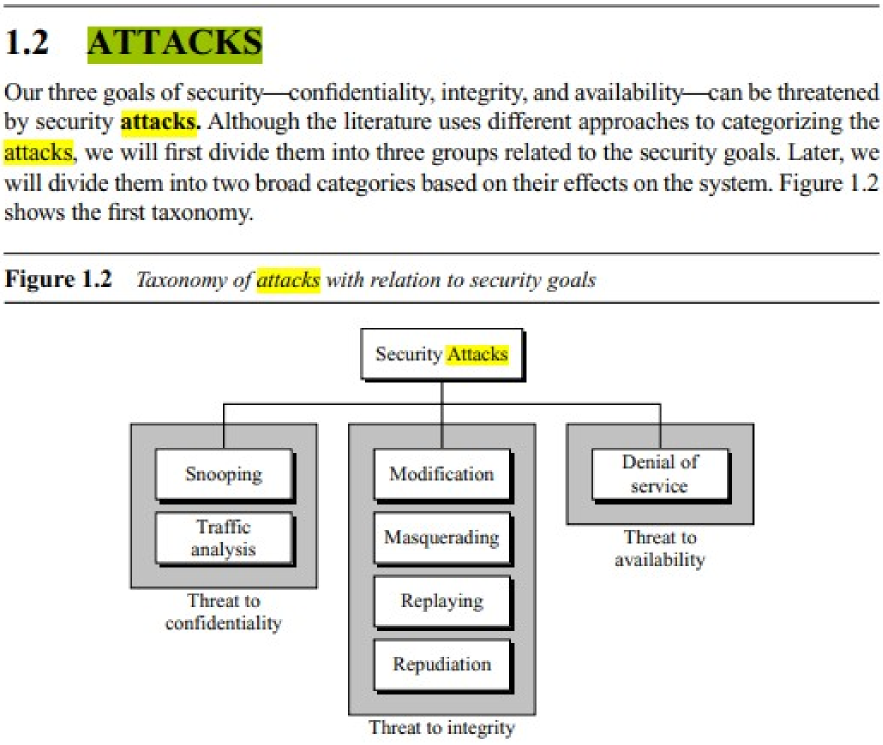


1.2 ATTACKS

Our three goals of security—confidentiality, integrity, and availability—can be threatened by security **attacks.** Although the literature uses different approaches to categorizing the attacks, we will first divide them into three groups related to the security goals. Later, we will divide them into two broad categories based on their effects on the system. Figure 1.2 shows the first taxonomy.

Figure 1.2 *Taxonomy of attacks with relation to security goals*

Attacks Threatening Confidentiality

In general, two types of attacks threaten the confidentiality of information: **snooping** and **traffic analysis.**

Snooping

Snooping refers to unauthorized access to or interception of data. For example, a file transferred through the Internet may contain confidential information. An unauthorized entity may intercept the transmission and use the contents for her own benefit. To prevent snooping, the data can be made nonintelligible to the intercepter by using encipherment techniques discussed in this book.

Traffic Analysis

Although encipherment of data may make it nonintelligible for the intercepter, she can obtain some other type information by monitoring online traffic. For example, she can find the electronic address (such as the e-mail address) of the sender or the receiver. She can collect pairs of requests and responses to help her guess the nature of transaction.

Attacks Threatening Integrity

The integrity of data can be threatened by several kinds of attacks: **modification, masquerading, replaying,** and **repudiation.**

Modification

After intercepting or accessing information, the attacker modifies the information to make it beneficial to herself. For example, a customer sends a message to a bank to do some transaction. The attacker intercepts the message and changes the type of transaction to benefit herself. Note that sometimes the attacker simply deletes or delays the message to harm the system or to benefit from it.

Masquerading

Masquerading, or spoofing, happens when the attacker impersonates somebody else. For example, an attacker might steal the bank card and PIN of a bank customer and pretend that she is that customer. Sometimes the attacker pretends instead to be the receiver entity. For example, a user tries to contact a bank, but another site pretends that it is the bank and obtains some information from the user.

Replaying

Replaying is another attack. The attacker obtains a copy of a message sent by a user and later tries to replay it. For example, a person sends a request to her bank to ask for payment to the attacker, who has done a job for her. The attacker intercepts the message and sends it again to receive another payment from the bank.

Repudiation

This type of attack is different from others because it is performed by one of the two parties in the communication: the sender or the receiver. The sender of the message might later deny that she has sent the message; the receiver of the message might later deny that he has received the message.

An example of denial by the sender would be a bank customer asking her bank to send some money to a third party but later denying that she has made such a request. An

---

3.4.2 How DES Works

1. Basic Principles

DES is a block cipher. It encrypts data in blocks of 64 bits each. That is, 64 bits of plain text goes as the input to DES, which produces 64 bits of cipher text. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is 56 bits. The basic idea is shown in Fig. 3.18.

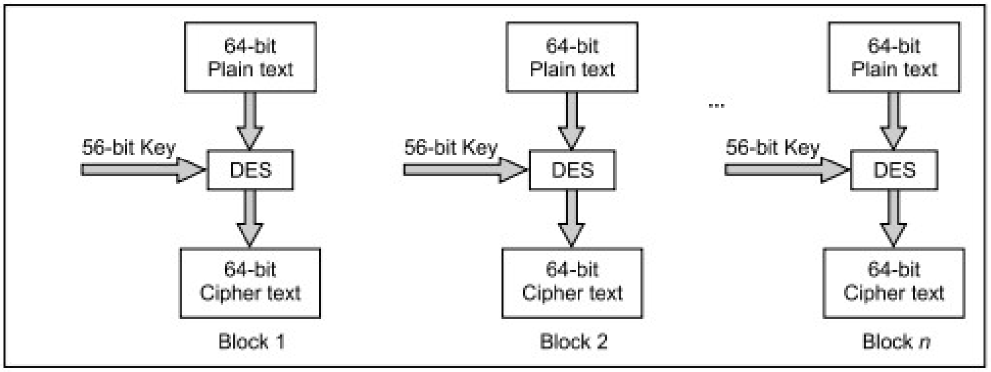


Fig. 3.18 Conceptual working of DES

We have mentioned that DES uses a 56-bit key. Actually, the initial key consists of 64 bits. However, before the DES process even starts, every eighth bit of the key is discarded to produce a 56-bit key. That is, bit positions 8, 16, 24, 32, 40, 48, 56 and 64 are discarded. This is shown in Fig. 3.19 with shaded bit positions indicating discarded bits. (Before discarding, these bits can be used for parity checking to ensure that the key does not contain any errors.)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64

Fig. 3.19 Discarding of every 8th bit of the original key (shaded bit positions are discarded)

Thus, the discarding of every 8th bit of the key produces a 56-bit key from the original 64-bit key, as shown in Fig. 3.20.

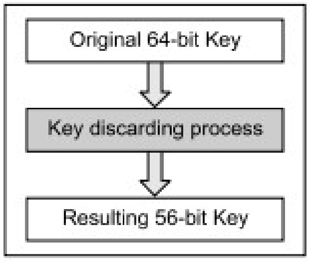


Fig. 3.20 Key-discarding process

Simplistically, DES is based on the two fundamental attributes of cryptography: substitution (also called confusion) and transposition (also called diffusion). DES consists of 16 steps, each of which is called a **round**. Each *round* performs the steps of substitution and transposition. Let us now discuss the broad-level steps in DES.

1. In the first step, the 64-bit plain-text block is handed over to an **Initial Permutation (IP)** function.
2. The initial permutation is performed on plain text.
3. Next, the Initial Permutation (IP) produces two halves of the permuted block; say Left Plain Text (LPT) and Right Plain Text (RPT).
4. Now, each of LPT and RPT go through 16 *rounds* of encryption process, each with its own key.
5. In the end, LPT and RPT are rejoined, and a **Final Permutation (FP)** is performed on the combined block.

Suppose that the cryptanalyst knows two basic pieces of information: *P* (a plain-text block), and *C* (the corresponding final cipher-text block) for a message. We know that the relations shown in Fig. 3.40 are true for *P* and *C*, if we are using double DES. The mathematical equivalents of these are also shown. The result of the first encryption is called *T*, and is denoted as $T = E_{K1}(P)$ [i.e. encrypt the block *P* with key *K*1]. After this encrypted block is encrypted with another key *K*2, we denote the result as $C = E_{K2}(E_{K1}(P))$ [i.e. encrypt the already encrypted block *T*, with a different key *K*2, and call the final cipher text as *C*].

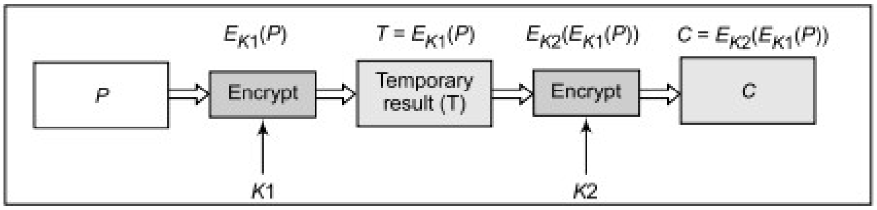


Fig. 3.40 Mathematical expression of double DES

Now, the aim of the cryptanalyst, who is armed with the knowledge of *P* and *C*, is to obtain the values of *K*1 and *K*2. What would the cryptanalyst do?

Step 1 For all possible values (2^{56}) of key *K*1, the cryptanalyst would use a large table in the memory of the computer, and perform the following two steps:

1. The cryptanalyst would encrypt the plain-text block *P* by performing the first encryption operation, i.e. $E_{K1}(P)$. That is, it will calculate *T*.
2. The cryptanalyst would store the output of the operation $E_{K1}(P)$, i.e. the temporary cipher text (*T*), in the next available row of the table in the memory.

We show this process for the ease of understanding using a 2-bit key (actually, the cryptanalyst has to do this using a 64-bit key, which makes the task a lot harder). Refer to Fig. 3.41.

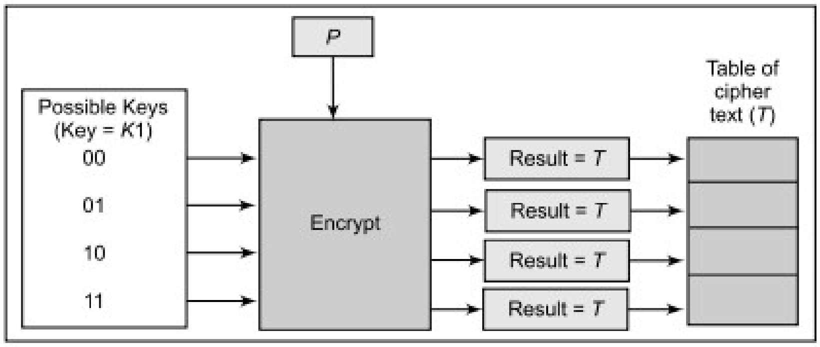
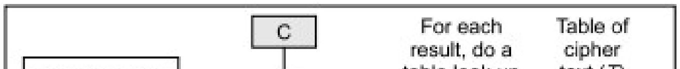
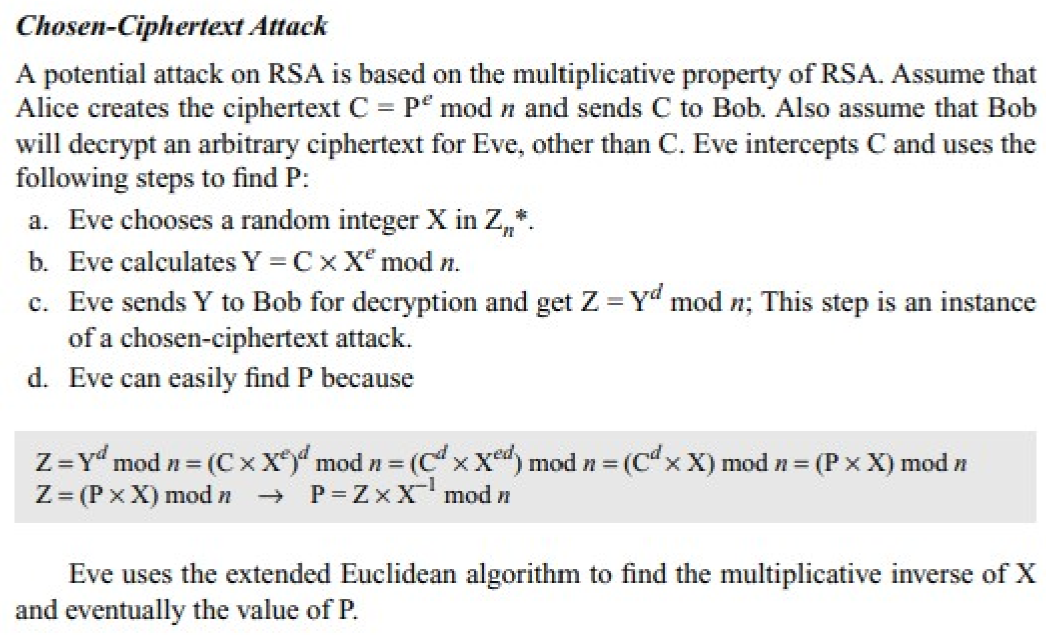


Fig. 3.41 Conceptual view of the cryptanalyst's *Encrypt* operation

Step 2 Thus, at the end of the above process, the cryptanalyst will have the table of cipher texts as shown in the figure. Next, the cryptanalyst will perform the reverse operation. That is, he/she will now decrypt the known cipher text *C* with all the possible values of *K*2 [i.e. perform $D_{K2}(C)$ for all possible values of *K*2]. In each case, the cryptanalyst will compare the resulting value with all the values in the table of cipher texts, which were computed earlier. This process (as before, for 2-bit key) is shown in Fig. 3.42.





Chosen-Ciphertext Attack

A potential attack on RSA is based on the multiplicative property of RSA. Assume that Alice creates the ciphertext $C = P^e \bmod n$ and sends C to Bob. Also assume that Bob will decrypt an arbitrary ciphertext for Eve, other than C. Eve intercepts C and uses the following steps to find P:

a. Eve chooses a random integer X in $Z_n{}^*$.
b. Eve calculates $Y = C \times X^e \bmod n$.
c. Eve sends Y to Bob for decryption and get $Z = Y^d \bmod n$; This step is an instance of a chosen-ciphertext attack.
d. Eve can easily find P because

$$Z = Y^d \bmod n = (C \times X^e)^d \bmod n = (C^d \times X^{ed}) \bmod n = (C^d \times X) \bmod n = (P \times X) \bmod n$$
$$Z = (P \times X) \bmod n \quad \rightarrow \quad P = Z \times X^{-1} \bmod n$$

Eve uses the extended Euclidean algorithm to find the multiplicative inverse of X and eventually the value of P.

Security of ElGamal

Two attacks have been mentioned for the ElGamal cryptosystem in the literature: attacks based on low modulus and known-plaintext attacks.

Low-Modulus Attacks

If the value of p is not large enough, Eve can use some efficient algorithms (see Chapter 9) to solve the discrete logarithm problem to find d or r. If p is small, Eve can easily find $d = \log_{e1} e_2 \bmod p$ and store it to decrypt any message sent to Bob.This can be done once and used as long as Bob uses the same keys. Eve can also use the value of C_1 to find random number r used by Alice in each transmission $r = \log_{e1} C_1 \bmod p$. Both of these cases emphasize that security of the ElGamal cryptosystem depends on the infeasibility of solving a discrete logarithm problem with a very large modulus. It is recommended that p be at least 1024 bits (300 decimal digits).

Known-Plaintext Attack

If Alice uses the same random exponent r, to encrypt two plaintexts P and P′, Eve discovers P′ if she knows P. Assume that $C_2 = P \times (e_2^r) \bmod p$ and $C'_2 = P' \times (e_2^r) \bmod p$. Eve finds P′ using the following steps:

1. $(e_2^r) = C_2 \times P^{-1} \bmod p$
2. $P' = C'_2 \times (e_2^r)^{-1} \bmod p$

It is recommended that Alice use a fresh value of r to thwart the known-plaintext attacks.

For the ElGamal cryptosystem to be secure, p must be at least 300 digits and r must be new for each encipherment.

Example 10.12

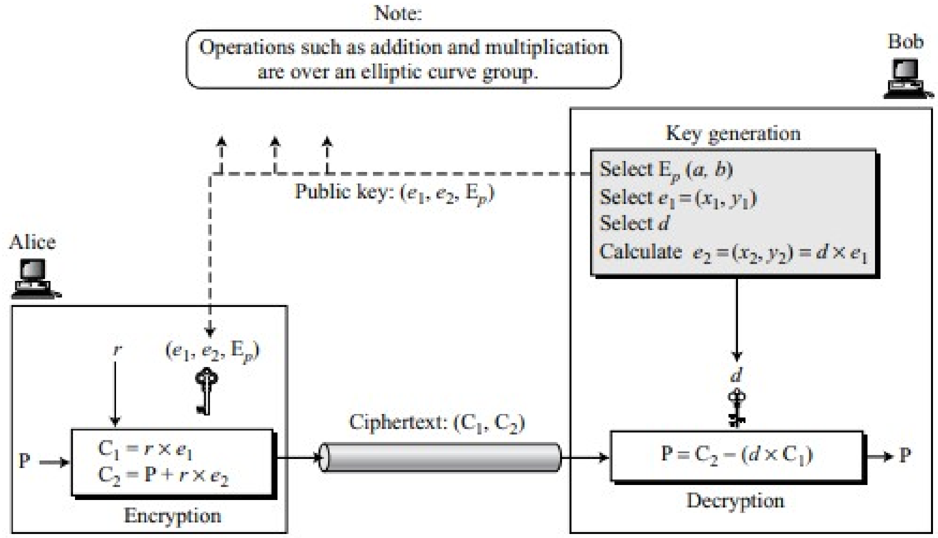
Here is a more realistic example. Bob uses a random integer of 512 bits (the ideal is 1024 bits). The integer p is a 155-digit number (the ideal is 300 digits). Bob then chooses e_1, d, and calculates e_2, as shown below: Bob announces (e_1, e_2, p) as his public key and keeps d as his private key.

$p =$	11534899272561676244925313717014331740490094532609834959814346921905689869862264593212975473787189514436889176526473093615929993728061165964347353440008577
$e_1 =$	2
$d =$	1007
$e_2 =$	9788641304300918950876685693809773904388006288733768761002206223325545070741561892123183177046101416733601508841329408572485377031582066010072558707455

Elliptic Curve Cryptography Simulating ElGamal

Several methods have been used to encrypt and decrypt using elliptic curves. The common one is to simulate the ElGamal cryptosystem using an elliptic curve over **GF**(p) or **GF**(2^n), as shown in Figure 10.16.

Figure 10.16 *ElGamal cryptosystem using the elliptic curve*



Generating Public and Private Keys

1. Bob chooses $E(a, b)$ with an elliptic curve over **GF**(p) or **GF**(2^n).
2. Bob chooses a point on the curve, $e_1(x_1, y_1)$.
3. Bob chooses an integer d.
4. Bob calculates $e_2(x_2, y_2) = d \times e_1(x_1, y_1)$. Note that multiplication here means multiple addition of points as defined before.
5. Bob announces $E(a, b)$, $e_1(x_1, y_1)$, and $e_2(x_2, y_2)$ as his public key; he keeps d as his private key.

Encryption

Alice selects P, a point on the curve, as her plaintext, P. She then calculates a pair of points on the text as ciphertexts:

$$\mathbf{C_1 = r \times e_1} \qquad\qquad \mathbf{C_2 = P + r \times e_2}$$

The reader may wonder how an arbitrary plaintext can be a point on the elliptic curve. This is one of the challenging issues in the use of the elliptic curve for simulation. Alice needs to use an algorithm to find a one-to-one correspondence between symbols (or a block of text) and the points on the curve.

Decryption

Comparison

The following shows a quick comparison of the original ElGamal algorithm with its simulation using the elliptic curve.

a. The original algorithm uses a multiplicative group; the simulation uses an elliptic group.
b. The two exponents in the original algorithm are numbers in the multiplicative group; the two multipliers in the simulation are points on the elliptic curve.
c. The private key in each algorithm is an integer.
d. The secret numbers chosen by Alice in each algorithm are integers.
e. The exponentiation in the original algorithm is replaced by the multiplication of a point by a constant.
f. The multiplication in the original algorithm is replaced by addition of points.
g. The inverse in the original algorithm is the multiplicative inverse in the multiplicative group; the inverse in the simulation is the additive inverse of a point on the curve.
h. Calculation is usually easier in the elliptic curve because multiplication is simpler than exponentiation, addition is simpler than multiplication, and finding the inverse is much simpler in the elliptic curve group than in a multiplicative group.

Security of ECC

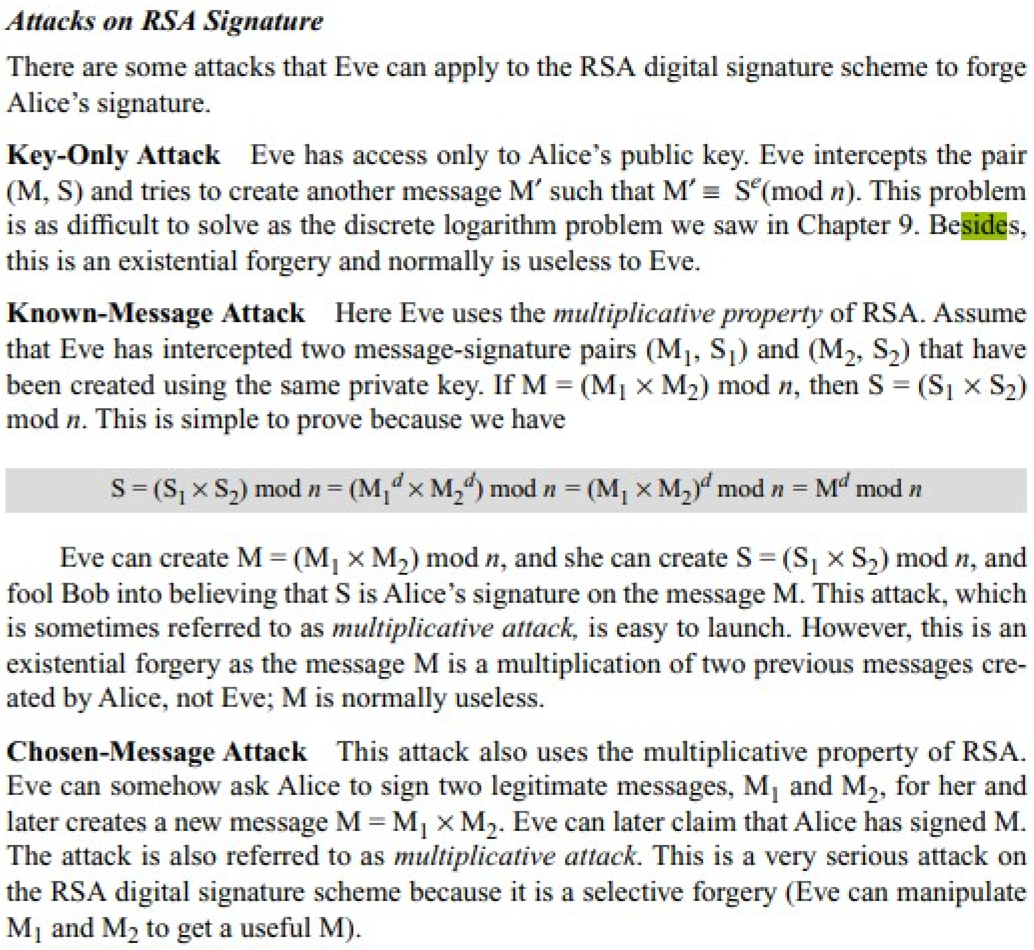
To decrypt the message, Eve needs to find the value of r or d.

a. If Eve knows r, she can use $P = C_2 - (r \times e_2)$ to find the point P related to the plaintext. But to find r, Eve needs to solve the equation $C_1 = r \times e_1$. This means, given two points on the curve, C_1 and e_1, Eve must find the multiplier that creates C_1 starting from e_1. This is referred to as the **elliptic curve logarithm problem,** and the only method available to solve it is the Polard rho algorithm, which is infeasible if r is large, and p in GF(p) or n in GF(2^n) is large.
b. If Eve knows d, she can use $P = C_2 - (d \times C_1)$ to find the point P related to the plaintext. Because $e_2 = d \times e_1$, this is the same type of problem. Eve knows the value of e_1 and e_2; she needs to find the multiplier d.

The security of ECC depends on the difficulty of solving the elliptic curve logarithm problem.

Modulus Size

For the same level of security (computational effort), the modulus, n, can be smaller in ECC than in RSA. For example, ECC over the **GF**(2^n) with n of 160 bits can provide the same level of security as RSA with n of 1024 bits.



Attacks on RSA Signature

There are some attacks that Eve can apply to the RSA digital signature scheme to forge Alice's signature.

Key-Only Attack Eve has access only to Alice's public key. Eve intercepts the pair (M, S) and tries to create another message M′ such that $M' \equiv S^e (\text{mod } n)$. This problem is as difficult to solve as the discrete logarithm problem we saw in Chapter 9. Besides, this is an existential forgery and normally is useless to Eve.

Known-Message Attack Here Eve uses the *multiplicative property* of RSA. Assume that Eve has intercepted two message-signature pairs (M_1, S_1) and (M_2, S_2) that have been created using the same private key. If $M = (M_1 \times M_2) \text{ mod } n$, then $S = (S_1 \times S_2) \text{ mod } n$. This is simple to prove because we have

$$S = (S_1 \times S_2) \text{ mod } n = (M_1{}^d \times M_2{}^d) \text{ mod } n = (M_1 \times M_2)^d \text{ mod } n = M^d \text{ mod } n$$

Eve can create $M = (M_1 \times M_2) \text{ mod } n$, and she can create $S = (S_1 \times S_2) \text{ mod } n$, and fool Bob into believing that S is Alice's signature on the message M. This attack, which is sometimes referred to as *multiplicative attack,* is easy to launch. However, this is an existential forgery as the message M is a multiplication of two previous messages created by Alice, not Eve; M is normally useless.

Chosen-Message Attack This attack also uses the multiplicative property of RSA. Eve can somehow ask Alice to sign two legitimate messages, M_1 and M_2, for her and later creates a new message $M = M_1 \times M_2$. Eve can later claim that Alice has signed M. The attack is also referred to as *multiplicative attack.* This is a very serious attack on the RSA digital signature scheme because it is a selective forgery (Eve can manipulate M_1 and M_2 to get a useful M).

Weaknesses in Cipher Design

We will briefly mention some weaknesses that have been found in the design of the cipher.

S-boxes At least three weaknesses are mentioned in the literature for S-boxes.

1. In S-box 4, the last three output bits can be derived in the same way as the first output bit by complementing some of the input bits.
2. Two specifically chosen inputs to an S-box array can create the same output.
3. It is possible to obtain the same output in a single round by changing bits in only three neighboring S-boxes.

P-boxes One mystery and one weakness were found in the design of P-boxes:

1. It is not clear why the designers of DES used the initial and final permutations; these have no security benefits.
2. In the expansion permutation (inside the function), the first and fourth bits of every 4-bit series are repeated.

Weakness in the Cipher Key

Several weaknesses have been found in the cipher key.

Key Size Critics believe that the most serious weakness of DES is in its key size (56 bits). To do a brute-force attack on a given ciphertext block, the adversary needs to check 2^{56} keys.

a. With available technology, it is possible to check one million keys per second. This means that we need more than two thousand years to do brute-force attacks on DES using only a computer with one processor.

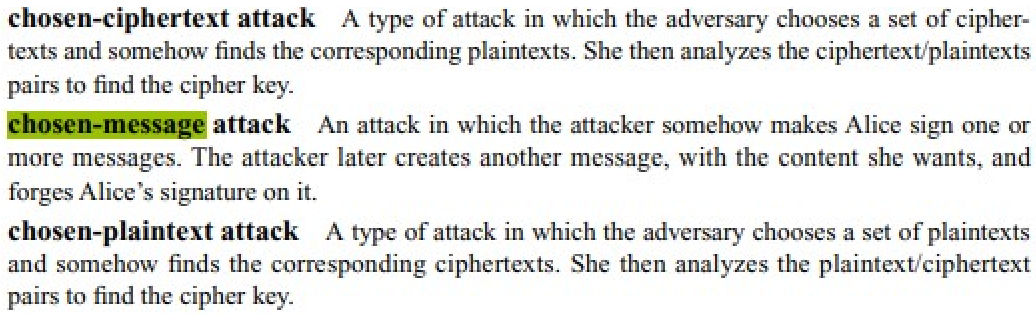
b. If we can make a computer with one million chips (parallel processing), then we can test the whole key domain in approximately 20 hours. When DES was introduced, the cost of such a computer was over several million dollars, but the cost has dropped rapidly. A special computer was built in 1998 that found the key in 112 hours.

c. Computer networks can simulate parallel processing. In 1977 a team of researchers used 3500 computers attached to the Internet to find a key challenged by RSA Laboratories in 120 days. The key domain was divided among all of these computers, and each computer was responsible to check the part of the domain.

d. If 3500 networked computers can find the key in 120 days, a secret society with 42,000 members can find the key in 10 days.

The above discussion shows that DES with a cipher key of 56 bits is not safe enough to be used comfortably. We will see later in the chapter that one solution is to use triple DES (3DES) with two keys (112 bits) or triple DES with three keys (168 bits).

Weak Keys Four out of 2^{56} possible keys are called **weak keys.** A weak key is the one that, after parity drop operation (using Table 6.12), consists either of all 0s, all 1s, or half 0s and half 1s. These keys are shown in Table 6.18.



chosen-ciphertext attack A type of attack in which the adversary chooses a set of ciphertexts and somehow finds the corresponding plaintexts. She then analyzes the ciphertext/plaintexts pairs to find the cipher key.

chosen-message attack An attack in which the attacker somehow makes Alice sign one or more messages. The attacker later creates another message, with the content she wants, and forges Alice's signature on it.

chosen-plaintext attack A type of attack in which the adversary chooses a set of plaintexts and somehow finds the corresponding ciphertexts. She then analyzes the plaintext/ciphertext pairs to find the cipher key.

4.7.7 Message Authentication Code (MAC)

The concept of **Message Authentication Code (MAC)** is quite similar to that of a message digest. However, there is one difference. As we have seen, a message digest is simply a fingerprint of a message. There is no cryptographic process involved in the case of message digests. In contrast, a MAC requires that the sender and the receiver should know a shared symmetric (secret) key, which is used in the preparation of the MAC. Thus, MAC involves cryptographic processing. Let us see how this works.

Let us assume that the sender *A* wants to send a message *M* to a receiver *B*. How the MAC processing works is shown in Fig. 4.46.

1. *A* and *B* share a symmetric (secret) key *K*, which is not known to anyone else. A calculates the MAC by applying key *K* to the message *M*.
2. A then sends the original message *M* and the MAC *H*1 to *B*.

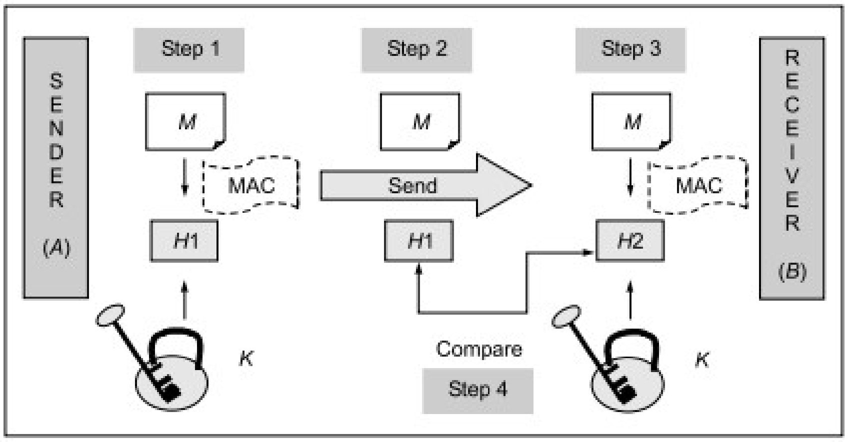


Fig. 4.46 Message Authentication Code (MAC)

3. When *B* receives the message, *B* also uses *K* to calculate its own MAC *H*2 over *M*.
4. *B* now compares *H*1 with *H*2. If the two match, *B* concludes that the message *M* has not been changed during transit. However, if *H*1 ≠ *H*2, *B* rejects the message, realizing that the message was changed during transit.

The significances of a MAC are as follows:

1. The MAC assures the receiver (in this case, *B*) that the message is not altered. This is because if an attacker alters the message but does not alter the MAC (in this case, *H*1) then the receiver's calculation of the MAC (in this case, *H*2) will differ from it. Why does the attacker then not also alter the MAC? Well, as we know, the key used in the calculation of the MAC (in this case, *K*) is assumed to be known only to the sender and the receiver (in this case, *A* and *B*). Therefore, the attacker does not know the key, *K*, and therefore, she cannot alter the MAC.
2. The receiver (in this case, *B*) is assured that the message indeed came from the correct sender (in this case, *A*). Since only the sender and the receiver (*A* and *B*, respectively, in this case) know the secret key (in this case, *K*), no one else could have calculated the MAC (in this case, H1) sent by the sender (in this case, *A*).

Interestingly, although the calculation of the MAC seems to be quite similar to an encryption process, it is actually different in one important respect. As we know, in symmetric-key cryptography, the cryptographic process must be reversible. That is, the encryption and decryption are the mirror images of each other. However, note that in the case of MAC, both the sender and the receiver are performing

---

4.7.8 HMAC

1. Introduction

HMAC stands for **Hash-based Message Authentication Code**. HMAC has been chosen as a mandatory security implementation for the Internet Protocol (IP) security, and is also used in the Secure Socket Layer (SSL) protocol, widely used on the Internet.

The fundamental idea behind HMAC is to reuse the existing message-digest algorithms, such as MD5 or SHA-1. Obviously, there is no point in reinventing the wheel. Therefore, what HMAC does it to work with any message-digest algorithm. That is, it treats the message digest as a black box. Additionally, it uses the shared symmetric key to encrypt the message digest, which produces the output MAC. This is shown in Fig. 4.47.

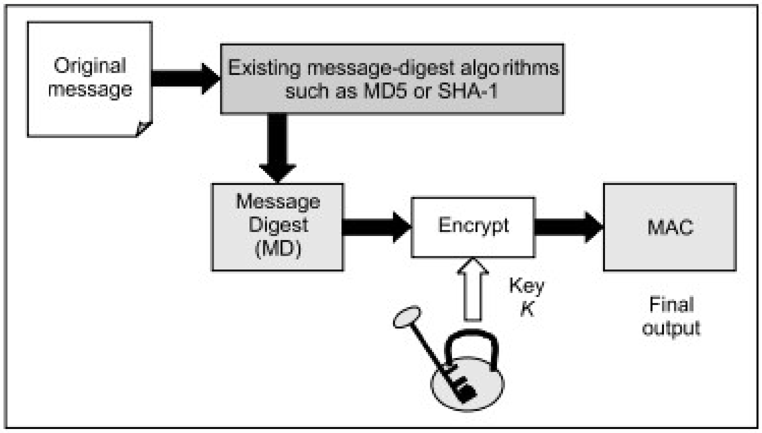


Fig. 4.47 HMAC concept

2. The Working of HMAC

Let us now take a look at the internal working of HMAC. For this, let us start with the various variables that will be used in our HMAC discussion.

MD = The message digest/hash function used (e.g. MD5, SHA-1, etc.)
M = The input message whose MAC is to be calculated
L = The number of blocks in the message M
b = The number of bits in each block
K = The shared symmetric key to be used in HMAC
ipad = A string 00110110 repeated b/8 times
opad = A string 01011010 repeated b/8 times

Armed with this input, we shall use a step-by-step approach to understand the HMAC operation.

Step 1: Make the length of *K* equal to *b* The algorithm starts with three possibilities, depending on the length of the key K:

- *Length of $K < b$* In this case, we need to expand the key (K) to make the length of K equal to the number of bits in the original message block (i.e. b). For this, we add as many 0 bits as required to

the left of K. For example, if the initial length of K = 170 bits, and b = 512 then we add 342 bits, all with a value 0, to the left of K. We shall continue to call this modified key as K.

- *Length of $K = b$* In this case, we do not take any action, and proceed to step 2.
- *Length of $K > b$* In this case, we need to trim K to make the length of K equal to the number of

---

2. Block Ciphers

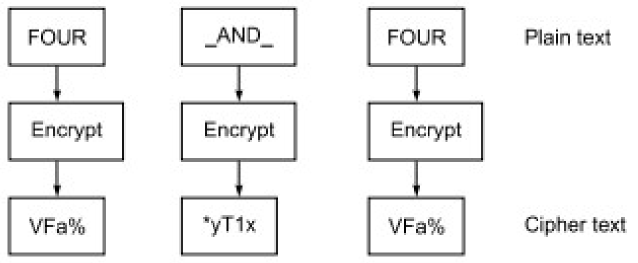
In **block ciphers**, rather than encrypting one bit at a time, a block of bits is encrypted at one go. Suppose we have a plain text *FOUR_AND_FOUR* that needs to be encrypted. Using block cipher, *FOUR* could be encrypted first, followed by *_AND_* and finally *FOUR*. Thus, one block of characters gets encrypted at a time.

During decryption, each block would be translated back to the original form. In actual practice, the communication takes place only in bits. Therefore, *FOUR* actually means the binary equivalent of the ASCII characters *FOUR*. After any algorithm encrypts these, the resultant bits are converted back into their ASCII equivalents. Therefore, we get funny symbols such as *Vfa%*, etc. In actual practice, their binary equivalents are received, which are decrypted back into binary equivalent of ASCII *FOUR*. This is shown in Fig. 3.4.

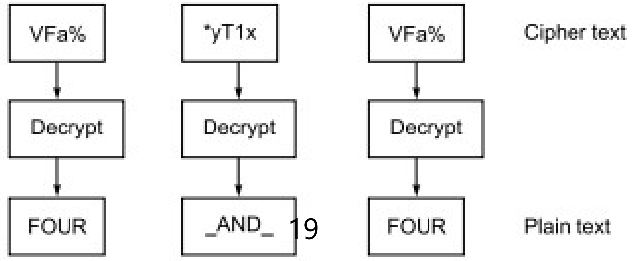
An obvious problem with block ciphers is repeating text. For repeating text patterns, the same cipher is generated. Therefore, it could give a clue to the cryptanalyst regarding the original plain text. The cryptanalyst can look for repeating strings and try to break them. If he/she succeeds in doing so, there is a danger that a relatively large portion of the original message is broken into, and therefore, the entire message can then be revealed with more effort. Even if the cryptanalyst cannot guess the remaining words, suppose he/she changes all *debit* to *credit* and vice versa in a funds transfer message, it could cause havoc! To deal with this problem, block ciphers are used in **chaining mode**, as we shall study later. In this approach, the previous block of cipher text is mixed with the current block, so as to obscure the cipher text, thus avoiding repeated patterns of blocks with the same contents.

> *The block-cipher technique involves encryption of one block of text at a time. Decryption also takes one block of encrypted text at a time.*

Practically, the blocks used in a block cipher generally contain 64 bits or more. As we have seen, stream ciphers encrypt one bit at a time. This can be very time-consuming and actually unnecessary in real



(a) The encryption process at the sender's end



(b) The decryption process at the receiver's end

Fig. 3.4 Block cipher

life. That is why block ciphers are used more often in computer-based cryptographic algorithms as compared to stream ciphers. Consequently, we will focus our attention on block ciphers with reference

---

3. *RSA and Digital Signatures*

We have mentioned that RSA can be used for performing digital signatures. Let us understand how this works in a step-by-step fashion. For this, let us assume that the sender (*A*) wants to send a message *M* to the receiver (*B*) along with the digital signature (*S*) calculated over the message (*M*).

Step 1 The sender (*A*) uses the SHA-1 message-digest algorithm to calculate the message digest (MD1) over the original message (*M*). This is shown in Fig. 4.56.

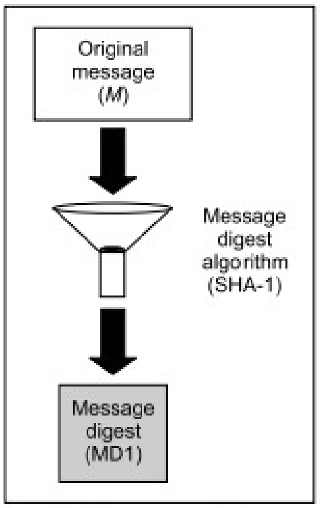


Fig. 4.56 Message-digest calculation

Step 2 The sender (*A*) now encrypts the message digest with her private key. The output of this process is called the digital signature (DS) of *A*. This is shown in Fig. 4.57.

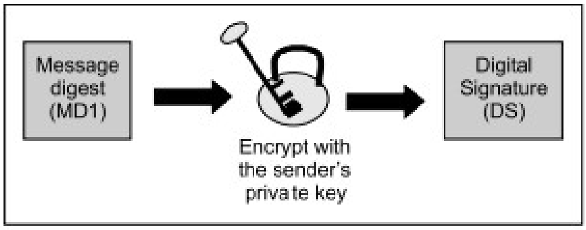


Fig. 4.57 Digital-signature creation

Step 3 Now the sender (*A*) sends the original message (*M*) along with the digital signature (DS) to the receiver (*B*). This is shown in Fig. 4.58.

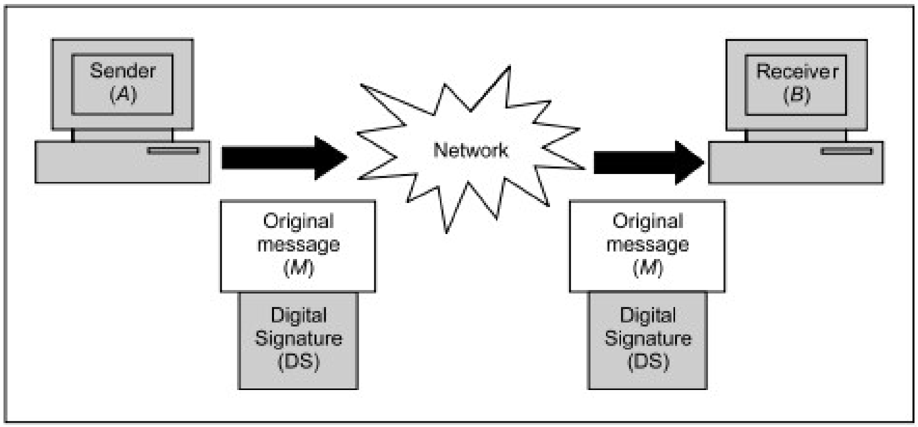
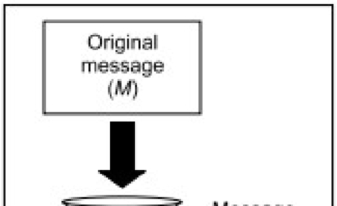


Fig. 4.58 Transmission of original message and digital signature simultaneously

Step 4 After the receiver (*B*) receives the original message (*M*) and the sender's (*A*'s) digital signature, *B* uses the same message-digest algorithm as was used by *A*, and calculates its own message digest (MD2) as shown in Fig. 4.59.

Step 5 The receiver (*B*) now uses the sender's (*A*'s) public key to decrypt (sometimes also called **de-sign**) the digital signature. Note that *A* had used her private key to encrypt her message digest (MD1) to form the digital signature. Therefore, only *A*'s public key can be used to decrypt it. The output of this process is the original message digest as was calculated by *A* (MD1) in step 1. This is



■ 4.9 ELGAMAL DIGITAL SIGNATURE ■

We have discussed the ElGamal cryptosystem earlier. The **ElGamal digital-signature** scheme uses the same keys, but a different algorithm. The algorithm creates two digital signatures. In the verification step, these two signatures are tallied. The key-generation process here is the same as what we had discussed earlier and hence we would not repeat the discussion. The public key remains ($E1$, $E2$, P) and the private key continues to be D.

4.9.1 Signature

The signature process works as follows:

1. The sender selects a random number R.
2. The sender computes the first signature $S1$ using the equation $S1 = E1^R \bmod P$.
3. The sender computes the second signature $S2$ using the equation $S2 = (M - D \times S1) \times R^{-1} \bmod (P - 1)$, where M is the original message that needs to be signed.
4. The sender sends M, $S1$, and $S2$ to the receiver.

For example, let $E1 = 10$, $E2 = 4$, $P = 19$, $M = 14$, $D = 16$, and $R = 5$.

Then we have:

$$S1 = E1^R \bmod P = 10^5 \bmod 19 = 3$$
$$S2 = (M - D \times S1) \times R^{-1} \bmod (P - 1) = (14 - 16 \times 3) \times 5^{-1} \bmod 18 = 4$$

Hence, the signature is ($S1$, $S2$) i.e. (3, 4). This is sent to the receiver.

4.9.2 Verification

The verification process works as follows:

1. The receiver performs the first part of verification called $V1$ using the equation $V1 = E1^M \bmod P$.
2. The receiver performs the second part of verification called as $V2$ using the equation $V2 = E2^{S1} \times S1^{S2} \bmod P$.

In our example:

$$V1 = E1^M \bmod P = 10^{14} \bmod 19 = 16$$
$$V2 = E2^{S1} \times S1^{S2} \bmod P = 4^3 \times 3^4 \bmod 19 = 5184 \bmod 19 = 16$$

Since $V1 = V2$, the signature is considered valid.

■ 4.10 ATTACKS ON DIGITAL SIGNATURES ■

In general, three types of attacks are attempted against digital signatures, as outlined below:

1. Chosen-message Attack

In the **chosen-message attack**, the attacker tricks a genuine user into digitally signing messages that the user does not normally intend to sign. As a result, the attacker obtains a pair of the original message

that was signed and the signature. Using this, the attacker tries to create a new message that she wants the genuine user to sign and uses the previous signature.

2. Known-message Attack

In the **known-message attack**, the attacker obtains the previous few messages and the corresponding digital signatures from a genuine user. Like the known-plain text attack in the case of encryption, the attacker now tries to create a new message and forge the digital signature of the genuine user onto it.

5.2.7 Certificate Hierarchies and Self-signed Digital Certificates

Another question that we might have at this stage is the **verification** of a digital certificate. In general, it looks quite satisfactory, except for one potential threat. Suppose that Alice has received Bob's digital certificate, and she wants to verify it. As we know, this means that Alice needs to de-sign the certificate using the CA's public key. Now, how does Alice know what is the CA's public key?

One possibility is that the CA of Alice and Bob is the same. In such a case, there is no problem, as Alice would already know the public key of the CA. However, this cannot always be guaranteed. For instance, Alice and Bob may not have obtained their certificates from the same CA. In such a case, how can Alice obtain the public key of the CA?

To resolve such problems, a **Certification Authority hierarchy** is created. This is also called the **chain of trust**. In simple terms, all the CAs are grouped into multiple levels of a CA hierarchy. This is shown in Fig. 5.20.

As the figure shows, the *Certification Authority (CA) hierarchy* begins with the **root CA**. The root CA has one or more second-level CAs below. Each of these second-level CAs can have one or more third-level CAs, which in turn can have lower level CAs, and so on. This is like a reporting hierarchy in an organization, where the CEO or the Managing Director is the highest authority. Many senior managers report to the CEO or the Managing Director. Many managers report to one senior manager. Many people would report to each manager, and so on.

What is the purpose of creating this hierarchy? Just as this sort of structure relieves the CEO or the Managing Director from performing all the possible tasks in all the departments, this sort of hierarchy relieves the root CA from having to manage all the possible digital certificates. Instead, the root CA can delegate this job to the second-level CAs. This delegation can happen region-wise (e.g. one second-level CA could be responsible for the Western region, another for the Eastern region, a third one for the Northern region, and a fourth one for the Southern region, etc.). Each of these second-level CAs could appoint third-level CAs state-wise within that region. Each third-level CA could delegate its responsibilities to a fourth-level CA city-wise, and so on.

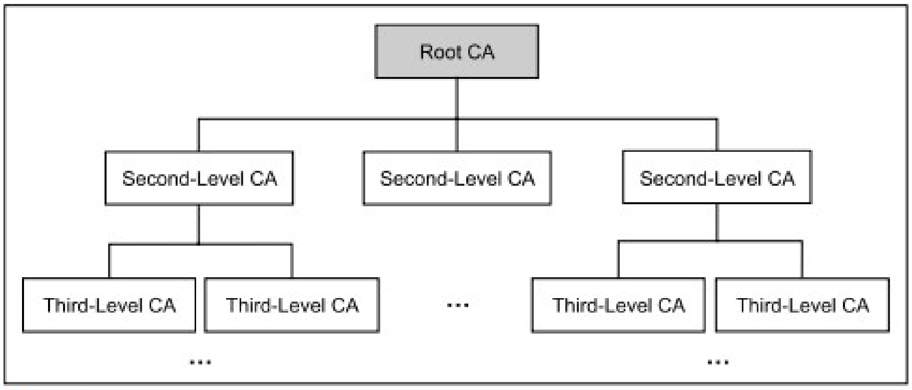
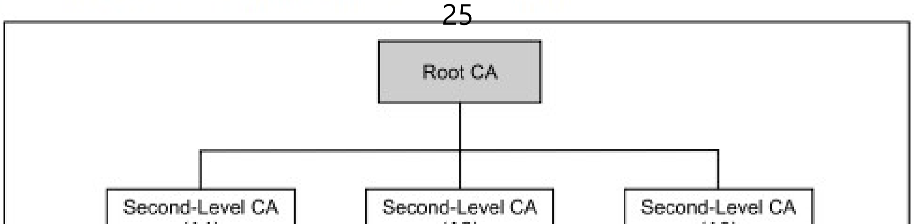


Fig. 5.20 CA hierarchy

Thus, in our example, if Alice has obtained her certificate from a third-level CA and Bob has obtained his certificate from a *different* third-level CA, how can Alice verify Bob's certificate? Let us understand this by naming the CAs for simplification, as shown in Fig. 5.21.



4.7 DIGITAL SIGNATURES

4.7.1 Introduction

All along, we have been talking of the following general scheme in the context of asymmetric-key cryptography:

> *If A is the sender of a message and B is the receiver, A encrypts the message with B's public key and sends the encrypted message to B.*

We have deliberately hidden the internals of this scheme. As we know, actually this is based on digital envelopes as discussed earlier, wherein not the entire message but only the one-time session key used to encrypt the message is encrypted with the receiver's public key. But for simplicity, we shall ignore this technical detail, and instead, assume that the whole message is encrypted with the receiver's public key.

Let us now consider another scheme, as follows:

> *If A is the sender of a message and B is the receiver, A encrypts the message with A's private key and sends the encrypted message to B.*

This is shown in Fig. 4.15.

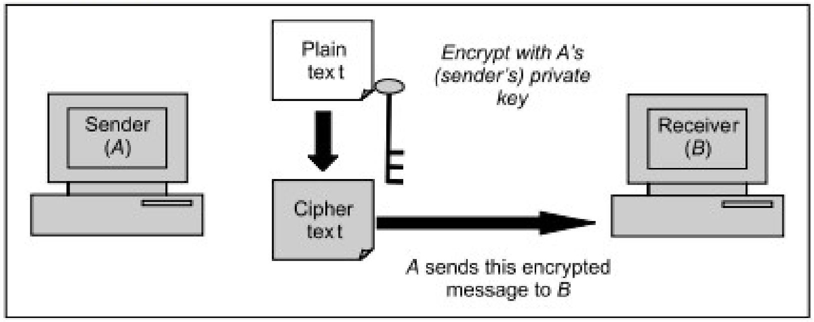


Fig. 4.15 Encrypting a message with the sender's private key

Our first reaction to this would be: what purpose would this serve? After all, *A*'s public key would be, well, *public*, i.e. accessible to anybody. This means that anybody who is interested in knowing the contents of the message sent by *A* to *B* can simply use *A*'s public key to decrypt the message, thus causing the failure of this encryption scheme!

Well, this is quite true. But here, when *A* encrypts the message with her private key, her intention is not to hide the contents of the message (i.e. not to achieve *confidentiality*), but it is something else. What can that intention be? If the receiver (*B*) receives such a message encrypted with *A*'s private key, *B* can use *A*'s public key to decrypt it, and therefore, access the plain text. Does this ring a bell? If the decryption is successful, it assures *B* that this message was indeed sent by *A*. This is because if *B* can decrypt a message with *A*'s public key, it means that the message must have been initially encrypted with *A*'s private key (remember that a message encrypted with a public key can be decrypted only with the corresponding private key, and vice versa). This is also because only *A* knows her private key. Therefore, someone posing as *A* (say *C*) could not have sent a message encrypted with *A*'s private key to *B*. *A* must have sent it. Therefore, although this scheme does not achieve confidentiality, it achieves *authentication* (identifying and proving *A* as the sender). Moreover, in the case of a dispute tomorrow, *B* can take the encrypted message, and decrypt it with *A*'s public key to prove that the message indeed came from *A*. This achieves the purpose of *non-repudiation* (i.e. *A* cannot refuse that she had sent this message, as the message was encrypted with her private key, which is supposed to be known only to her).

Even if someone (say *C*) manages to intercept and access the encrypted message while it is in transit, then uses *A*'s public key to decrypt the message, changes the message, that would still not achieve any

---